IEICE TRANS. INF. & SYST., VOL.E88–D, NO.3 MARCH 2005

PAPER

Delay Fault Testing of Processor Cores in Functional Mode

Virendra SINGH[†a)], ***Nonmember***, **Michiko INOUE**[†b)], ***Member***, **Kewal K. SALUJA**[††c)], ***Nonmember***, ***and*** **Hideo FUJIWARA**[†d)], ***Fellow***

SUMMARY This paper proposes an efficient methodology of delay fault testing of processor cores using their instruction sets. These test vectors can be applied in the functional mode of operation, hence, self-testing of processor core becomes possible for path delay fault testing. The proposed approach uses a graph theoretic model (represented as an Instruction Execution Graph) of the datapath and a finite state machine model of the controller for the elimination of functionally untestable paths at the early stage without looking into the circuit details and extraction of constraints for the paths that can potentially be tested. Parwan and DLX processors are used to demonstrate the effectiveness of our method.

key words: *processor test, delay fault testing, software-based self-test, at-speed test*

1. Introduction

Aggressive processor design methods using giga-hertz clock and deep sub-micron technology are necessitating the use of at speed testing of these processors. It is no longer sufficient to target stuck-at faults, since other faults, such as delay faults, are becoming increasingly important to test. At-speed testing using an external tester is not an economically viable option. Self-test is an alternative solution. Widely used self-test technique, Built-In Self-Test (BIST), is a structural testing methodology that provides good quality tests but requires additional hardware and hence it has area and performance overhead. Further, this method may be unacceptable for testing an optimized processor core that is embedded deep inside a System-on-a-Chip (SoC). Once a core is embedded then it is also difficult to access the core for test application. Also, to use BIST, the circuit must be BIST ready, which often requires design changes.

Structural BIST operates in orthogonal test mode with pseudorandom test data, which may lead to excessive power dissipation. For at speed testing complex issues related to timing, like multiple clock domain and clock skew, must be resolved. In order to solve these problems, software-based self-test technique is an alternative. Software based self-test (functional self-test) methodology uses processor instructions and its functionality in order to test the processor core. Therefore, it has the following advantages over structural BIST: 1) it operates in normal mode of operation, 2) there is no area and performance overhead, 3) at speed testing is possible, and 4) there is no excessive power consumption during test.

Functional self-test can be easily applied on a processor core, embedded deep inside an SoC. This paper focuses on functional self-test of processor cores. We propose a delay fault testing methodology using functional self-test. We treat controller and datapath differently as both have different characteristics. Controller provides constraints on application of test vectors on itself and on the datapath. We first extract the constraints on datapath and controller and use these constraints in the test vector generation process. As the vectors are generated under constraints, instruction(s) to apply the test vectors can always be found.

The paper is organized as follows. Section 2 describes previous work in software based self-test and Sect. 3 describes the overview of our work and definitions used. Sections 4 and 5 describe test methodology for datapath and controller respectively. Section 6 discusses test instruction sequence generation. Section 7 describes the experimental results and finally the paper concludes with Sect. 8.

2. Previous Work

A number of software based self-test approaches [3]–[8], targeting stuck-at faults, have been proposed. The approaches proposed in [3] and [4] are based on instruction randomization and give low fault coverage due to high level of abstraction. In [5] the concept of self-test signature is introduced which is used to generate pseudorandom test patterns during test application. Due to pseudorandom nature of this methodology self-test code size and test application time are large. A deterministic self-test methodology is presented in [6], [7]. Deterministic nature of this approach leads to reduced test code size but is unable to achieve high fault coverage for complex architectures. A scalable methodology based on test program templates is presented in [8] which uses statistical regression method for function mapping. Approaches in [5]–[8] do not explicitly consider faults in the controller.

A software based self-test approach targeting delay faults is proposed in [9]–[11]. This approach, first classifies a path to be functionally testable or untestable by ex-

Manuscript received June 3, 2004.
Manuscript revised August 28, 2004.

[†]The authors are with the Nara Institute of Science and Technology (NAIST), Ikoma-shi, 630–0192 Japan.
[††]The author is with the University of Wisconsin-Madison, USA.
a) E-mail: virend-s@is.naist.jp, viren@ceeri.res.in
b) E-mail: kounoe@is.naist.jp
c) E-mail: saluja@engr.wisc.edu
d) E-mail: fujiwara@is.naist.jp

DOI: 10.1093/ietisy/e88–d.3.610

tracting a set of constraints for the datapath logic and the controller. The authors argue that delay defects on the functionally untestable paths will not cause any chip failure. In constraint extraction procedure for datapath, all instruction pairs are enumerated and for each instruction pair all possible vector pairs that can be applied to the datapath are derived symbolically. This requires a substantial effort to analyze all the instructions and all possible pairs of instructions even though it is not necessary to analyze all the pairs as shown in this paper. Path classification procedure in controller uses sequential path classification methodology i.e., in order to classify a path it propagates the transition forward to Primary Outputs (PO) of the controller and backward to Primary Inputs (PI) of the controller in multiple time frames under the constraints. The method proposed in this paper extracts the constraints on state transitions, eliminating the need for consideration of multiple time frames. Results on controller are not reported in [9]–[11]. Based on the above approach, a delay fault testing methodology of an SoC using its own embedded processor instructions is presented in [12].

Our methodology uses graph theoretic model of datapath (represented by Instruction Execution Graph) and finite state machine model of the controller to eliminate the functionally untestable paths at the early stage without considering circuit details. This eliminates a substantial number of functionally untestable faults. Our approach is different from the methods proposed in [9]–[11] which consider circuit details for path classification and multiple time frames for the controller.

3. Overview of Proposed Work

Our methodology considers datapath and controller separately as both of these have different design characteristics. The activities in the datapath are controlled by the controller. The controller is also constrained by state transitions and signals from the datapath. Hence, only a subset of structurally applicable test vectors may be applied in the functional mode. Path delay fault model is used.

We model datapath by a new graph theoretic model called Instruction Execution Graph (IE-Graph) that can be constructed from the instruction set architecture and RT level description of the processor. In our formulation of the test problem IE-Graph is used to classify all paths as *functionally untestable paths* (FUTP) or *potentially functionally testable paths* (PFTP), and to extract the constraints imposed on the datapath for PFTPs. First, constraints on the control signals that can be applied on the paths between a pair of registers in consecutive cycles are extracted. Next, constraints on justifiable data inputs (registers) are extracted. Following these, a combinational constrained Automatic Test Pattern Generator (ATPG) is used to generate test vectors under the extracted constraints. Thus, in this approach only those vectors are generated that can be applied functionally. Further, the search space is significantly small as only those states are used during test generation which can cause data transfer to take place on a path between a pair of registers.

For testing the controller, the constraints are extracted in the form of state transitions from its RT level description. These constraints also include the values of status signals in the status register and instruction code in the instruction register of the processor. After extracting the constraints, paths are classified as FUTP or PFTP. Combinational constrained ATPG is used to generate the test vectors for the paths classified as PFTP. As the vectors must be generated with constraints on the states and inputs to the controller (contents of the instruction register and status register), the number of time frames that are required for sequential test generation are eliminated. In the final phase, test instructions are generated using the knowledge of the control signals and contents of the instruction register. Justification and observation instruction sequence generation processes are based on heuristics which minimize the number of instructions and/or the test application time. In order to test the processor core, the test program (a sequence of generated instructions) is loaded into the memory. The processor fetches the instructions from the memory and after execution, the results are transferred to the memory.

Throughout this paper the following concepts and notation will be used.

Definition 1: A *path* [15] is defined as an ordered set of gates $\{g_0, g_1, \ldots, g_n\}$, where g_0 is a primary input or output from a FF, and g_n is a primary output or input to a FF. Output of a gate g_i is an input to gate g_{i+1} $(0 < i < n - 1)$.

Definition 2: A path is (enhanced-scan or standard-scan) *structurally testable* [9] if there exists a structural test for the path, which can be applied through the (enhanced or standard) full-scan chain.

Definition 3: A path is *functionally testable* [9] if there exists a functional test for that path, otherwise the path is functionally untestable.

A functional two-pattern test does not exist to test a path implies that there does not exist an instruction or an instruction sequence to apply the required test in functional mode of operation. Clearly, functionally untestable paths are never activated in normal (functional) operational mode and we need not target these paths in our approach. We use the following notation to represent signal values.

- c: represents a bit which has the same value as in the previous timeframe.
- x: represents a bit that can be assigned either a logic 0 or a logic 1 value at will.
- d: represents a bit which is not cared by state transition. It is the same as x, except that legitimate bit pattern in the register has to be justified.
- R: represents rising transition.
- F: represents falling transition.

Definition 4: A constraint P is represented by a vector pair and each element of P can be 0, 1, x, c, or d.

Table 1 Instruction set of Parwan processor.

1. LDA	5. JMP	9. BRA_C	13. CLA	17. ASR
2. AND	6. STA	10. BRA_Z	14. CMA	
3. ADD	7. JSR	11. BRA_N	15. CMC	
4. SUB	8. BRA_V	12. NOP	16. ASL	

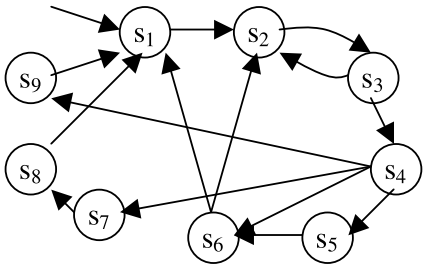

Fig. 1 State transition diagram of Parwan processor.

Definition 5: A constraint P is said to *cover* a constraint Q if P = Q or Q can be obtained from P by assigning 0 and 1 values to x's in P.

We will use Parwan processor [14] as a running example in this paper to explain many of the concepts. Parwan processor is an accumulator based 8-bit processor with a 12-bit address bus. It has 17 instructions, listed in Table 1, and it supports both direct and indirect addressing modes. The state diagram of the controller in the Parwan processor is shown in Fig. 1.

4. Datapath

In this section we deal with only those paths that are relevant to data transfer between registers in the datapath. The paths which include the logic in the controller are considered in the next section, even if they start from and end at some registers in the datapath.

Datapath is modeled by an IE-Graph. This is based on the concept of S-Graph proposed in [1], [2]. However, unlike S-Graph, the IE-Graph contains information about data transfer activities associated with an instruction as well as the state during which a given action takes place. IE-Graph is constructed from the instruction set architecture and register transfer level description and includes architecture registers of the datapath.

Nodes of the IE-Graph are: i) registers, ii) two special nodes, IN and OUT, which model external world such as memory and I/O devices, iii) part of registers which can be independently readable and writable, and iv) equivalent registers (set of registers which behave in the same way with instruction set, as defined by [2]), such as registers in a register file. A directed edge between two nodes is drawn iff there exists at least one instruction which is responsible for transfering data (with or without manipulation) over the paths between two nodes (registers). Each edge is marked with a set of [state, instruction(s)] pairs, which are responsible for the data transfer between the pair of nodes. Partial IE-Graph of Parwan processor is shown in Fig. 2. Complete graph is given in [20].

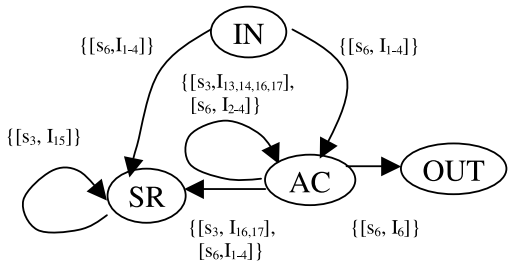

Fig. 2 Partial IE-graph of Parwan processor.

Test vector generation process uses instruction set architecture, RT level description, and gate level netlist. It is a two-step process. The first step is constraint extraction process and the second step is test vector generation process.

4.1 Constraint Extraction and Path Classification

There are two types of constraints imposed on the datapath by the controller: i) control constraints, and ii) data constraints. Control constraints are imposed on control signals, which are responsible for transferring data between two registers. These constraints are obtained from IE-Graph and RT level description. Data constraints, on the other hand, are the constraints on the justifiable data in the registers under control constraints, which are obtained from RT level description.

Definition 6: Let there be an edge from node R_i to R_o, marked with $[s_l, I_p]$. The marked state s_l is defined as a *latching state* for the paths represented by that edge.

Data transfer activity from register R_i to R_o takes place in state s_l during the execution of instruction I_p and register R_o will be latched. Hence, state s_l is defined as a latching state.

Lemma 1: Let $<V_1, V_2>$ be a test vector pair for a path from register R_i to a register R_o, where test vector V_1 is followed by V_2 and the edge between these registers is marked with a set of state-instruction pairs $\{[s_l, I_p]\}$. This vector pair can be a test vector pair in functional mode only if there exists at least one state-instruction pair $[s_l, I_p] \in \{[s_l, I_p]\}$, such that

1. vector V_2 can be applied in the latching state s_l of the instruction I_p, and
2. vector V_1 can be applied in the state just before the latching state s_l of the instruction I_p.

Note that every instruction is a sequence of state transitions and the latching state(s) in this sequence for a register pair is well defined. However, if the latching state happens to be the very first state of an instruction then the last state of every instruction needs to be considered as the state that immediately precedes the latching state.

During the latching state s_l, data transfer (with or without manipulation) from register R_i to R_o takes place and the

result is latched in register R_o. Therefore, we can apply the second vector only in the latching state (say s_l) and the first vector must be applied in a state just before the latching state (say s_j). Two consecutive states s_j and s_l provide the control constraints, and control signals in these states during the execution of instruction(s) marked with the latching state are obtained from RT level description. Constraints on the states during which we can apply the test vectors $<V_1, V_2>$ take care of justification of the control signals in the functional mode of testing. Data constraints in the form of justifiable data in the input register of the register pair and other registers required for the execution of marked instruction are obtained from RT level description.

Lemma 2: Paths from register R_i to R_o are *functionally untestable* if the following conditions exist,

1. R_i is not an IN node, and
2. R_i has no incoming edge marked with the state just before the latching state (s_l) of the instruction Ip for any $[s_l, I_p]$ marked on the edge (R_i, R_o).

If conditions stated in Lemma 2 exist then transition cannot be launched from register R_i. Hence, the paths between a register pair R_i and R_o are FUTP. Otherwise, these paths are classified as PFTPs and we need to extract the data constraints for the these paths. Covering relation, defined in Sect. 2, helps reduce the number of constraints.

Following examples should help clear the above concepts.

Example 1: Constraints on paths between AC and AC:
The edge between nodes AC and AC is marked with $\{[s_3, (I_{13-14}, I_{16-17})], [s_6, I_{2-4}]\}$, as shown in Fig. 2. The previous states of s_3 and s_6 are s_2 and s_4(or s_5), respectively (as shown in Fig. 1). AC is neither an IN node nor it has any incoming edge which is marked with just previous state of its latching state s_3 or s_6. Therefore, using Lemma 2 we can conclude that paths from AC to AC are functionally untestable.

Example 2: Paths from IN to AC:
The edge between nodes IN and AC is marked with $[s_6, I_{1-4}]$, as shown in Fig. 2. These paths are PFTP in accordance with Lemma 2, as input node is an IN node, and the latching state for these paths is s_6. Therefore, control constraints are the control signals generated in state s_4 or s_5 followed by s_6 for the instructions I_1, I_2, I_3 or I_4. This is obtained from IE-Graph and RT level description. The states s_4 and s_5 are the previous state of the latching state s_6 (as shown in Fig. 1). Data constraints can be obtained in the state s_4 followed by state s_6 or state s_5 followed by s_6, for the instructions I_1, I_2, I_3 and I_4. Data constraints for the instruction I_3 are shown in Table 2. This table shows the constraints for ALU control signals (ALU ctrl) and SHU control signals (SHU ctrl) as control constraints and constraints for IN and AC as data constraints in consecutive two time frames corresponding to s_4 and s_6 states of the instruction I_3. We consider these constraints as ALU and SHU lie in the paths and AC is the other input needed by the instruction I_3. The extracted data constrained for IN and AC shows that any value can be justified in IN, where as AC must have the same value across two time frames. Here we assume that when input to a combinational logic is in high impedance state then it can hold the logic value that is applied before the high impedance state. Parwan processor uses tristate buses which are responsible for the constraints on IN node.

Table 2 Data constraints for the paths between IN and AC.

State	I_3(ADD)			
	ALU ctrl	SHU ctrl	IN	AC (other i/p)
s_4	000	00	xxxx_xxxx	xxxx_xxxx
s_6	101	00	xxxx_xxxx	cccc_cccc

```
Constraint Extraction Procedure
1.Constraint path pair set W = Φ
2.for nodes R_i (i = 1 to n) {  // there are  n nodes in IE-Graph
3.    for  each edge  (R_i, R_j) (j = 1 to m) { //
                  // there are m edges from node R_i //
4.          if paths are PFTP then {   // (using Lemma 2) //
5.                  P_ij = Set of all paths between R_i and R_j
6.                  C_ij= Set of constraints for the paths from
                          node R_i to node R_j
7.                  W = W ∪ {[ C_ij, P_ij ]}
8.          }
9.       }
10.}
Test Generation Procedure
   Constrained ATPG  process
   Input : Constraint path pair set W, Gate level net list
   Output : Set of testable path  with their test vector pairs
```

Fig. 3 Constraint extraction and test generation procedures.

For instruction I_3, both control constraints, s_4 followed by s_6 and s_5 followed by s_6, are identical. Hence, using the covering relation one of these two constraints can be eliminated. All other constraints are extracted similarly.

4.2 Test Vector Generation Procedure

Constrained ATPG is used to generate the test vectors for the PFTPs under the extracted constraints. Path lists between a register pair and their corresponding constraints are provided as inputs to an ATPG along with gate level netlist and it returns the test vectors for the testable path. Procedure to extract the constraints and test generation is given in Fig. 3. This procedure systematically extracts the constraints using IE-Graph and uses constrained ATPG to generate the test vectors.

5. Controller

Controller is a sequential circuit that is normally implemented as a Mealy type or a Moore type finite state machine. Structural organization of the controller is shown in Fig. 4.

In this section, we treat all paths which include logic elements in the controller. Test vectors applicable in functional mode of operation to the controller are restricted by the state transitions. If for a path there exists no sequence of valid state transitions which can launch a transition and propagate it along the path then that path is a functionally untestable path, even though that may be structurally testable. Therefore, we extract constraints on state transitions prior to test generation.

5.1 Constraint Extraction

Change of state of controller is determined by the contents of registers (IR and SR), inputs (PI), and the present state. Input from registers IR and SR (i.e., registers other than the present state register (PSR)) are treated as Constrained Primary Input (CPI). Therefore, we need to extract two types of constraints: i) constraints on state transition, and ii) constraint on legitimate values in IR and SR registers, because these are treated as constrained primary input.

1. Constraints on state transitions:
   Constraints on state transitions can be extracted by extracting possible valid state transition under legitimate values in IR, SR and input, by using instruction set architecture and RT level description. We demonstrate this using Parwan processor as an example. Table 3 shows a part of the state transition table of Parwan processor.
   This table shows that when present state is s_1 then next state will be either s_1 or s_2 depending on the value of input, and independent of values in IR and SR registers.

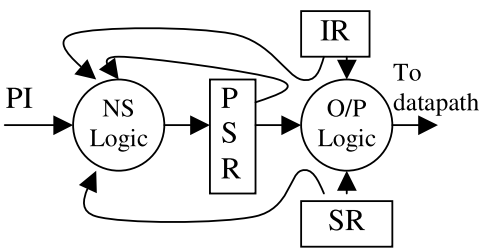

Fig. 4 Structural organization of the controller.

Table 3 State transition table of Parwan processor. (Partial)

PS	NS	IR (PS)	IR (NS)	SR (PS)	SR (NS)	In (PS)
s_1	s_1	dddd_dddd	cccc_cccc	dddd	cccc	1
	s_2	dddd_dddd	cccc_cccc	dddd	cccc	0
s_2	s_3	dddd_dddd	0xxx_xxxx	dddd	cccc	d
			100x_xxxx	dddd	cccc	d
			101x_xxxx	dddd	cccc	d
			110x_xxxx	dddd	cccc	d
			1110_0000	dddd	cccc	d
			1110_0001	dddd	cccc	d
			1110_0010	dddd	cccc	d
			1110_0100	dddd	cccc	d
			1110_1000	dddd	cccc	d
			1110_1001	dddd	cccc	d

   During these state transitions (s_1 to s_1 or s_2) register IR and SR can have any legitimate value in the present state (s_1) and must have the same values in next state (s_1 or s_2). Hence, we cannot launch transition from IR and SR during these state transitions. When present state is s_2 then next state is always s_3. IR can have any legitimate value in present state (s_2) as well as in next state (s_3). Therefore, transition can be launched from IR during the state transition s_2 to s_3.
2. Constraints on legitimate values in IR and SR registers (registers other than the present state register):
   A set of legitimate values in the registers other than the present state register can be obtained from its instruction set architecture and RT level description. For example, the legitimate bit patterns that the register IR of the Parwan processor can have are specified as {IR, < 0xxx_xxxx, 10xx_xxxx, 110x_xxxx, 1111_0100, 1111_0010, 1111_0001, 1110_0000, 1110_0001, 1110_0010, 1110_0100, 1110_1000, 1110_1001>}.

5.2 Path Classification

After extraction of constraints, each path is classified as PFTP or FUTP. This process uses state transition diagram and gate level implementation. There are three types of paths in a controller

1. PSR to PSR
2. PI or CPI (registers IR and SR) to PSR
3. PI, CPI, or PSR to a register in datapath.

Paths from PSR to PSR are only responsible for sequential behavior of the controller circuit. For path classification, we construct a table that shows transition on bits in PSR and other registers with state transitions. Table 4 shows transition on bits in PSR with state transitions for Parwan processor when states are binary encoded.

This table shows that there can be rising transition on bit b_3 only when there is a state transition from state s_4 to s_9. Similarly, there can be falling transition on b_3 only when there is a state transition from state s_9 to s_1.

Lemma 3: Paths between bit i in register R_1 and bit j in register R_2 (registers R_1 and R_2 need not be different) in controller circuit are functionally untestable paths for a transition (rising or falling) if

Table 4 Transition on bits in PSR with state transition. (Parwan processor)

bit		s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
b_3	R				s_9					
	F									s_1
b_2	R				s_5,s_6,s_7					
	F						s_1,s_2		s_1	
b_1	R		s_3							
	F				s_5,s_6,s_9					
b_0	R	s_2		s_4		s_6		s_8		
	F		s_3		s_5,s_7,s_9		s_1		s_1	

1. there does not exist a valid state transition s_m to s_n to launch the transition at bit i, or
2. there does not exist a state transition s_p to s_q which can receive the launched transition (launched during the state transition s_m to s_n) or its inverse (receive falling transition when rising is launched) at bit j, such that $s_n = s_p$.

A path is functionally testable if we can create a transition and propagate its effect along the path. If the conditions stated in Lemma 3 exist then we either cannot launch a transition or cannot propagate the created transition. Hence, the paths between bit i in register R_1 and bit j in register R_2 are FUTPs. Otherwise, these paths are classified as PFTPs because transitions can be created and may be propagated if values in other registers are justifiable. We also get precise constraints under which these paths can be tested using state transition table.

1. Paths from PSR to PSR (Paths from controller to controller):
   A path between bit i and bit j in PSR can be classified as follows for rising transition, using Lemma 3. We consider 3 consecutive time frames as shown in the following table. Activities at bit i and j in PSR and required state transitions are listed in Table 5. These paths are PFTP iff either s_m to s_n to s_r, or s_m to s_p to s_q state transition sequence exists.

   Example 3: PSR to PSR paths classification in Parwan processor when states are binary encoded. Table 4 shows transition on bits in PSR with state transitions. Paths from b_2 to b_1 (for rising transition) are classified as FUTP because no one state transition sequence exists to test these paths, where as paths from b_2 to b_1 (for falling transition) are classified as a PFTP because a state transition sequence s_6 to s_2 to s_3 exists. State transition sequence s_6 to s_2 to s_3 is an exact constraint for these paths (b_2 to b_1, falling transition) under which these can be tested if other values are justifiable. Similarly, we can find out all PFTPs, which are the potential candidates for the next phase.

2. Paths from PI or CPI to bit i in PSR (Paths from input or datapath to controller):
   a. Paths from PI to bit i in PSR are classified as PFTP, iff there exists a state sequence (s_m to s_n), which can receive a transition at bit i.
   b. Paths from CPI to bit i in PSR are classified as PFTP, iff there exist a valid state transition (s_m to s_n) to create a transition at CPI (register IR or SR) and there exists a valid state transition (s_n to s_p) at bit i of PSR (according to Lemma 3).
3. Paths from CPI or PSR to a register in datapath (Paths from controller to datapath):
   a. there exists a state sequence (s_m to s_n) which can launch a transition at bit i in PSR or CPI, and
   b. the register in the datapath, where these paths terminate, has an incoming edge, marked with state sn in IE-Graph and state s_m and s_n are two consecutive states of the marked instruction I_p.

Table 5 Activities at bit i and j in PSR.

Time frame	k	$k+1$	$k+2$
bit i	0	1	x
bit j	x	0 (1)	1 (0)
state	s_m	$s_n(s_p)$	$s_r(s_q)$

5.3 Test Vector Generation

A constrained combinational ATPG is used to generate the test vectors for the paths, which are, classified as PFTPs under the extracted constraints. ATPG is given with a set of PFTP along with their respective constraints. ATPG will return the test vectors if a path is testable under constraints.

This approach extracts the constraints in the form of state transitions and classifies the paths as functionally untestable or potentially functionally testable. Functionally untestable paths are removed from the path list. It uses combinational constraint ATPG to generate test vectors. Therefore, we need not consider multiple time frames for all the paths like a sequential ATPG, as sequential behavior is taken care of by the state transition in our approach. This also reduces the complexity of test generation. Note that the vectors generated by us are valid instructions and/or data.

6. Test Instruction Sequence Generation

The generated test vector pairs as explained in the preceding section are assigned to control signals and registers. Control signals and value(s) in IR in two consecutive time frames give the test instruction(s). Data in registers and in memory, which will be used by the test instruction, must be justified, using justification instruction. The result from the output register must be transferred to memory using observation instructions.

For example, consider a test vector pair (V_1, V_2), where V_1 = {ALU ctrl=000, SHU ctrl=00, AC=48H, IN=24H}, V_2 = {ALU ctrl=111, SHU ctrl=00, AC=48H, IN=04H}.

Figure 5 shows the generated test instructions for this example. In this figure, each line shows the memory location followed by an instruction mnemonic (with comments) or a content of the location. The generated test implies that a SUB instruction (which provides ALU ctrl = 000, SHU

```
Instructions
000H  LDA 400H   -- Load value from 400H to AC
002H  SUB 424H   -- AC <= Value at 424 –AC
004H  STA 401H   -- Store AC to 401H
Data
400H  48H
424H  04H
```

Fig. 5 Instruction sequence.

ctrl = 00 at s_4, and ALU ctrl = 111, SHU ctrl = 00 at s_6) is applied as a test instruction.

The lower order 8 bit of the memory address used by the SUB instruction (i.e., 424H) must be 24H (value of IN in V_1) and the value stored at this location must be 04H (value of IN in V_2). The value of AC is 48H in V_1. This implies that we should justify the value at AC prior to SUB instruction. This is achieved by LDA instruction that loads 48H to AC. The test instruction SUB stores the result in AC. We can observe the result by STA instruction (store AC) that transfers the result to memory.

We can generate a test instruction sequence for every test vector pair in the above stated manner.

7. Experimental Results

A constraint extraction procedure for the datapath and the controller has been implemented in C language. Constrained ATPG for delay fault testing has also been implemented in C language, as commercially available ATPGs are not capable of handling our constraints.

We have applied our methodology to Parwan processor [14] and DLX processor [16]. The synthesized version of the Parwan processor contains 888 gates and 53 flip-flops and DLX processor contains 16152 gates and 1446 flip-flops. IE-Graphs for these processors are constructed and functionally untestable paths are identified. For example, the paths from AC to AC, AC to SR, AC to OUT, SR to SR, SR to OUT and PC to PC in the datapath of the Parwan processor are found to be untestable. We extract constraints for rest of the paths using IE-Graph and RT level description. Similarly, state transition tables for both of the processors are constructed which show the constraints on the controllers.

We generated test patterns for Robust (Rob), Non Robust (NR) [15] and Functional Sensitizable (FS) [15] paths under the extracted constraints using our constrained ATPG. The test vectors are generated in the following order: Rob, NR, and FS. For each path, first it generates a test vector pair for the robust test, if exist under the constraints; otherwise go for NR test followed by FS test. Here we consider, the paths which are starting from some register in datapath (e.g., IR or SR) going through the controller and terminating at some register in datapath, as a part of the controller. These paths are large in number (about 98–99% of the total paths in the controller). Results are shown in the Tables 6 and 7.

The results show that 37% of paths in Parwan datapath, 46% paths in Parwan controller, and about 17% paths in DLX datapath are identified as functionally untestable paths and these are eliminated during the first phase without using circuit details. However, all the paths in the controller of the DLX processor are determined to be potentially functionally testable paths because of the completely specified state space in the state encoding (64 states are encoded in 6 bits). We have extracted the constraints for these paths efficiently and achieved 100% fault efficiency. The paths shown as functionally untestable in 9^{th} row of the Tables 6 and 7 include the paths which are declared functionally untestable in first phase. NR testable paths include the robust testable paths, and FS testable paths include Robust and NR testable paths. The CPU time shown in Tables 6 and 7 is the total

Table 6 Results for Parwan processor.

	Datapath			Controller		
	Rob	NR	FS	Rob	NR	FS
Total Path	5,217	5,217	5,217	174,362	174,362	174,362
No. of faults	10,434	10,434	10,434	348,724	348,724	348,724
Faults declared untestable in first phase	3,902	3,902	3,902	162,812	162,812	162,812
Percentage of eliminated faults	37.40	37.40	37.40	46.69	46.69	46.69
No. of functionally testable faults	156	1,653	2,618	407	2,417	3,520
Fault coverage (%)	1.50	15.84	25.09	0.12	0.69	1.01
No. of unctionally untestable faults	10,278	8,781	7,816	348,3173	346,307	345,204
Fault efficiency (%)	100	100	100	100	100	100
CPU time (ATPG)	3 minutes 41 sec			3 hours 27 minutes		

Table 7 Results for DLX processor.

	Datapath			Controller		
	Rob	NR	FS	Rob	NR	FS
Total Path	264,906	264,906	264,906	743,411	743,411	743,411
No. of faults	529,812	529,812	529,812	1468,822	1468,822	1468,822
Faults declared untestable in first phase	89,848	89,848	89,848	0	0	0
Percentage of eliminated faults	16.95	16.95	16.95	0	0	0
No. of functionally testable faults	19,354	34,924	44,324	15,146	42,295	112,735
Fault coverage (%)	3.65	6.59	8.37	1.03	2.88	7.68
No. of unctionally untestable faults	510,458	494,888	485,488	1453,676	1426,527	1356,087
Fault efficiency (%)	100	100	100	100	100	100
CPU time (ATPG)	1 hour 32 minutes			15 hours 18 minutes		

Table 8 Comparison with earleir work. (Fault Coverage)

		Parwan		DLX	
		Lai [10] work	Our work	Lai [10] work	Our work
Datapath	Rob	–	1.5	–	3.6
	NR	3.7	15.8	7.2	6.5
	FS	–	25.0	–	8.3
Controller	Rob	–	0.1	–	1.0
	NR	–	0.7	–	2.8
	FS	–	1.0	–	7.6

time taken by ATPG to generate Rob, NR and FS test vectors.

Table 8 shows that our methodology achieves higher fault coverage for the datapath of Parwan processor as compared to [10] because we are considering at microinstruction level during the extraction of constraints for the potentially testable paths. Note that [10] uses a different synthesized version of Parwan processor with 168 sequential elements in order to separate out controller and datapath to make it better testable and reduction of number of paths, where as we are using original Parwan processor. Similarly, their DLX processor is also differently synthesized. The results for the controller are not shown in [10]. Our approach can eliminate substantial number of paths without looking into the circuit details, where as [10] uses the circuit details for path classification. Moreover, [10] has not shown the results other the results for NR test for the datapath of Parwan and DLX processors.

Results show that very small number of the paths are functionally testable. Our approach can be extended for pipelined architecture by considering pipeline registers in IE-Graph.

8. Conclusion

A systematic approach for the delay fault testing of processor core using its instruction set has been presented in this paper. A graph theoretic model for data path has been developed. This model is used with the RT level description to eliminate the functionally untestable paths at the early stage and it is also used for extraction of constraints. Controller is modeled as a finite state machine and constraints on state transitions are extracted. This eliminates the need for multiple time frame consideration for test generation, and hence reduces the test generation complexity. Our experimental results show that our test generation process can efficiently generate test vectors for functionally testable paths which can be applied by test instructions.

Acknowledge

This work was supported in part by Semiconductor Technology Academic Research Center (STARC) under the Research Project and in part by Japan Society for the Promotion of Science (JSPS) under Grants-in-Aid for Scientific Research B (2) (No. 15300018).

References

[1] S.M. Thatte and J.A. Abraham, "Test generation for microprocessors," IEEE Trans. Comput., vol.C-29, no.6, pp.429–441, June 1980.
[2] D. Brahme and J.A. Abraham, "Functional testing of microprocessors," IEEE Trans. Comput., vol.33, no.6, pp.475–484, June 1984.
[3] J. Shen and J.A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," Proc. International Test Conference 1998, pp.990–999, 1998.
[4] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," Proc. VLSI Test Symposium 1999, pp.34–40, 1999.
[5] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.20, no.3, pp.369–380, March 2001.
[6] N. Krantis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Instruction-based self-testing of processor cores," Proc. VLSI Test Symposium 2002, pp.223–228, 2002.
[7] N. Krantis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-based self-testing of processor cores," J. Electron. Test., Theory Appl. (JETTA), vol.19, pp.103–112, 2003.
[8] L. Chen, S. Ravi, A. Raghunath, and S. Dey, "A scalable software-based self-test methodology for programmable processors," Proc. Design Automation Conference 2003, pp.548–553, ACM Press, 2003.
[9] W.-C. Lai, A. Krstic, and K.-T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," Proc. VLSI Test Symposium 2000, pp.15–20, 2000.
[10] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test program synthesis for path delay faults in microprocessor cores," Proc. International Test Conference 2000, pp.1080–1089, 2000.
[11] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Functionally testable path delay faults on a microprocessor," IEEE Des. Test Comput., vol.17, no.4, pp.6–14, Oct-Dec. 2000.
[12] W.-C. Lai and K.-T. Cheng, "Instruction-level DFT for testing processor and IP cores in system-on-a-chip," Proc. Design Automation Conference 2001, pp.59–64, ACM Press, NY, 2001.
[13] A. Krstic, L. Chen, W.-C. Lai, K.-T. Cheng, and S. Dey, "Embedded software-based self-test for programmable core-based designs," IEEE Des. Test Comput., vol.19, no.4, pp.18–27, July-Aug. 2002.
[14] Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, New York, 1997.
[15] A. Krstic and K.-T. Cheng, Delay fault testing for VLSI circuits, Kluwer Academic Publishers, 1998.
[16] J.L. Hennesy and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, 1996.
[17] M. Gumm, VLSI Design Course: VHDL-Modelling and Synthesis of the DLXS RISC Processor, University of Stuttgart, Germany, 1995.
[18] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Software-based delay fault testing of processor cores," Proc. Asian Test Symposium 2003, pp.68–71, 2003.
[19] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Instruction-based delay fault testing of processor cores," Proc. International Conference on VLSI Design 2004, pp.933–938, 2004.
[20] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Software-based delay fault testing of processor cores," NAIST Technical Report, NAIST-IS-TR2003006, May 2003.
http://isw3.aist-nara.ac.jp/IS/TechReport/report/2003006.pdf

Virendra Singh obtained his B.E and M.E degrees in Electronics and Communication Engineering from Malaviya National Institute of Technology, Jaipur, India, in 1994 and 1996 respectively. He is currently with the Central Electronics Engineering Research Institute (CEERI), Pilani, India, as a Scientist since March 1997. Prior to joining this, he served as Assistant Professor at Banasthali University, India from June 1996 to March 1997. Presently, he is on study leave from CEERI and has been pursuing Ph.D. at Nara Institute of Science and Technology, Nara, Japan. At CEERI, he has worked on design and development of high power microwave devices, Gigabit network design, and application specific instruction set processor design. His research interests are design validation and test of high performance processors, fault-tolerant computing, and network processor design. He is a member of IEEE, VLSI Society of India (VSI), and life member of the Institute of Electronics and Telecommunication Engineers (IETE) of India.

Michiko Inoue received her B.E., M.E, and Ph.D. degrees in computer science from Osaka University in 1987, 1989, and 1995 respectively. She worked at Fujitsu Laboratories Ltd. from 1989 to 1991. She is an associate professor of Graduate School of Information Science, Nara institute of Science and Technology (NAIST). Her research interests include distributed algorithms, parallel algorithms, graph theory and design and test of digital systems. She is a member of IEEE, the Information Processing Society of Japan (IPSJ), and Japanese Society for Artificial Intelligence.

Kewal K. Saluja obtained his Bachelor of Engineering (BE) degree in Electrical Engineering from the University of Roorkee, India in 1967, MS and Ph.D. degrees in Electrical and Computer Engineering from the University of Iowa, Iowa City in 1972 and 1973 respectively. He is currently with the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison as a Professor, where he teaches courses in logic design, computer architecture, microprocessor based systems, VLSI design and testing, and fault-tolerant computing. Prior to this he was at the University of Newcastle, Australia. Professor Saluja has held visiting and consulting positions at various national and international institutions includingUniversity of Southern California, Hiroshima University, Nara Institute of Science and Technology, and the University of Roorkee. He has also served as a consultant to the United Nations Development Program. He was the general chair of the 29th FTCS and he served as an Editor of the IEEE Transactions on Computers (1997–2001). He is currently the Associate Editor for the letters section of the Journal of Electronic Testing: Theory and Applications (JETTA). Professor Saluja is a member of Eta Kappa Nu, Tau Beta Pi, a fellow of the JSPS and a Fellow of the IEEE.

Hideo Fujiwara received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1969, 1971, and 1974, respectively. He was with Osaka University from 1974 to 1985 and Meiji University from 1985 to 1993, and joined Nara Institute of Science and Technology in 1993. In 1981 he was a Visiting Research Assistant Professor at the University of Waterloo, and in 1984 he was a Visiting Associate Professor at McGill University, Canada. Presently he is a Professor at the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. His research interests are logic design, digital systems design and test, VLSI CAD and fault tolerant computing, including high-level/logic synthesis for testability, test synthesis, design for testability, built-in self-test, test pattern generation, parallel processing, and computational complexity. He is the author of Logic Testing and Design for Testability (MIT Press, 1985). He received the IECE Young Engineer Award in 1977, IEEE Computer Society Certificate of Appreciation Award in 1991, 2000 and 2001, Okawa Prize for Publication in 1994, IEEE Computer Society Meritorious Service Award in 1996, and IEEE Computer Society Outstanding Contribution Award in 2001. He is an advisory member of IEICE Trans. on Information and Systems and an editor of IEEE Trans. on Computers, J. Electronic Testing, J. Circuits, Systems and Computers, J. VLSI Design and others. Dr. Fujiwara is a fellow of the IEEE, a Golden Core member of the IEEE Computer Society, and a fellow of the Information Processing Society of Japan.